Fine-grained Language Composition: A Case Study*

Edd Barrett[1], Carl Friedrich Bolz[2], Lukas Diekmann[3], and Laurence Tratt[4]

1 **Software Development Team, Department of Informatics, King's College London.** http://soft-dev.org/ http://eddbarrett.co.uk/
2 **Software Development Team, Department of Informatics, King's College London.** http://soft-dev.org/ http://cfbolz.de/
3 **Software Development Team, Department of Informatics, King's College London.** http://soft-dev.org/ http://lukasdiekmann.com/
4 **Software Development Team, Department of Informatics, King's College London.** http://soft-dev.org/ http://tratt.net/laurie/

Abstract

Although run-time language composition is common, it normally takes the form of a crude Foreign Function Interface (FFI). While useful, such compositions tend to be coarse-grained and slow. In this paper we introduce a novel fine-grained syntactic composition of PHP and Python which allows users to embed each language inside the other, including referencing variables across languages. This composition raises novel design and implementation challenges. We show that good solutions can be found to the design challenges; and that the resulting implementation imposes an acceptable performance overhead of, at most, 2.6x.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases JIT, tracing, language composition

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.3

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.1

1 Introduction

Language composition allows programmers to create systems written in a mix of programming languages. Most commonly, a Foreign Function Interface (FFI) to C is provided so that programs can interact with external libraries. However, other instances of language composition are rare, as crossing the barrier between arbitrary languages is difficult. In many cases, the only way to do so is by having different languages run their parts of the system in separate processes that communicate using (slow) inter-process communication mechanisms. The most common alternative is to use a single Virtual Machine (VM) (e.g. a Java VM), and translate all languages into that VM's bytecode format. This enables finer-grained compositions, but their performance is still generally underwhelming [3].

We believe that there are two different types of friction which make good language compositions difficult: *semantic friction* occurs when an aspect of one language has no

* This research was funded by the EPSRC COOLER (EP/K01790X/1) and LECTURE (EP/L02344X/1) grants.

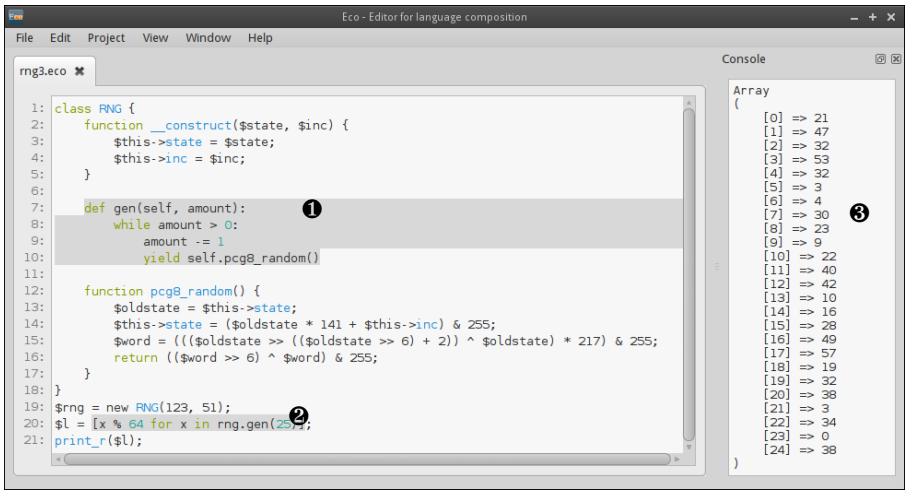

Figure 1 A PCG8 pseudo-random number generator [25] PyHyp program, written in the Eco language composition editor. In this case, the composed PHP and Python program will be exported to PyHyp compatible source code. The outer (white background) parts of the file are written in PHP, the inner (grey background) parts of the file in Python. ❶ A Python language box is used to add a generator method written in Python to the PHP class `RNG`. ❷ A Python language box is used to embed a Python expression inside PHP, including a cross-language variable reference for `rng` (defined in line 19 in PHP and referenced in line 20 in Python). In this case, a Python list comprehension builds a list of random numbers. When the list is passed to PHP, it is 'adapted' as a PHP array. ❸ Running the program pretty-prints the adapted Python list as a PHP array.

equivalent in the other; and *performance friction* occurs when the implementation of one language's behaviour forces the other to execute slowly.

Our hypothesis is that it is possible to reduce the currently accepted levels of friction in language compositions. We believe that the only way to test this hypothesis is through a concrete case study, since friction manifests in different ways in each language composition. We therefore composed together two real-world, widely used languages, Python and PHP, to make a new language composition called PyHyp. At a low-level, PyHyp defines a (somewhat traditional) FFI between PHP and Python that allows cross-language calls and the exchange of data. Building on the FFI, PyHyp then provides the basis for a novel syntactic composition. As shown in Figure 1, a single file can contain multiple fragments of PHP and Python code, and variables can be referenced across different language fragments (e.g. Python code can 'see' PHP variables and vice versa). Unlike approaches which translate one language into another, PyHyp does not alter existing language semantics, nor does it limit users to a subset of either language.

Depending on how one chooses to classify programming languages, Python and PHP can appear similar—most obviously, both are dynamically typed. From our perspective, however, there are a number of tricky differences: PHP has multiple global namespaces which can span multiple files, whereas Python uses one global namespace per file (semantic friction); most of PHP's core data-structures are immutable, whereas many of Python's are mutable (semantic and performance friction); and PHP's sole collection data type is a mapping, whereas Python separates the notion of mappings from that of sequences (semantic and performance friction). As this may suggest, this combination of languages presents a number of design and implementation challenges which have no obvious precedent. We show that PyHyp's solutions to these challenges allow interesting case studies to be implemented.

The practicality of our work rests on two recent developments. First, and most important, is the concept of interpreter composition. The basic idea is to make use of systems which can generate Just-In-Time (JIT) compiled VMs solely from the description of an interpreter (e.g. RPython [6] or Truffle [32]). There are three existing compositions in this style: Prolog and Python [3]; C and Ruby [16]; and C, Ruby, and JavaScript [17]. In essence, each of these systems implements a traditional FFI between its constituent interpreters. All of the systems have good peak performance, but implement simple FFIs. PyHyp defines a much finer-grained FFI between its two languages and also enables syntactic composition between the two languages. The second concept we make use of is language boxes as found in the Eco editor [11], which allow users to naturally write fragments of different languages alongside each other. Note that PyHyp neither extends, nor requires, Eco; however, Eco does hide several tedious details from users.

Although universal answers to our hypothesis are impossible, PyHyp shows that it is possible to create compositions which validate our hypothesis. To summarise, we show that:

1. PyHyp's FFI addresses a number of challenging semantic friction points.
2. Syntactic composition is possible and that practical designs can be found for novel cross-language features.
3. PyHyp's fine-grained language composition has, in the worst case, a performance overhead of 2.6x over its mono-language constituents.

A VirtualBox VM containing repeatable experiments, data and case studies is available at `http://dx.doi.org/10.4230/DARTS.2.1.1`.

2 Background

We assume a basic knowledge of Python syntax and semantics, but not of meta-tracing, interpreter composition, or PHP. In this section we provide overviews of the latter three.

2.1 Meta-tracing

Tracing JIT compilers record hot loops ('traces') in an interpreted program, optimise those traces, and then compile them into machine code [2, 13]. An individual trace is thus a record of one particular path through a program's control flow graph. Subsequent executions which follow that same path can use the machine code generated from the trace instead of the (slow) interpreter. To ensure that the path followed really is the same, 'guards' are left in the machine code at every possible point of divergence. If a guard fails, execution then reverts back to the interpreter.

Meta-tracing JITs have the same basic model, but replace the manually written tracer and machine code generator with equivalents automatically generated from the interpreter itself [24, 29, 33, 4, 6]. The key to good meta-tracing performance is heavily optimised traces. Language implementers can annotate the interpreter provide 'hints' to the meta-tracer to improve the quality of compiled traces. For example, hints can be used to mark parts of the interpreter constant, allowing the trace optimiser to apply constant folding. Similarly, hints can mark a function in the interpreter as 'elidable': given the same inputs, it always returns the same outputs.[1] The optimiser can then replace calls to (slow) elidable functions with

[1] Unlike pure functions, elidable functions can have idempotent side-effects (e.g. caching). The user is responsible for guaranteeing that the relationship between inputs and outputs is maintained.

(fast) checks on input values, substituting the output values in place of the function call. Identifying parts of an interpreter amenable to such hints requires the author's knowledge of both the semantics of the language being implemented and common idioms of use.

In this paper we use RPython, the main extant meta-tracing language. RPython is a statically typed subset of Python with a type system similar to Java's. Unlike seemingly similar languages (e.g. Slang [18] or PreScheme [22]), RPython is more than just a thin layer over C: it is, for example, fully garbage collected and has several high-level data types (e.g. lists and dictionaries). Despite this, VMs written in RPython have performance levels which far exceed traditional interpreters [6]. The specific details of RPython are generally unimportant in most of this paper, and we do not dwell on them: we believe that one could substitute any reasonable meta-tracing language (or its cousin approach, self-optimising interpreters with dynamic partial evaluation [32]) and achieve similar results.

2.2 Interpreter Composition

Interpreter composition involves 'glueing' together two or more existing interpreters such that each can utilise the other. Assuming that both interpreters are written in the same language, a basic composition is simple: interpreter A needs to import interpreter B and then call appropriate functionality in B. A more sophisticated composition will define matters such as data type conversion, and how and when execution passes between its constituent interpreters. Achieving the desired composition can require adding entirely new glue code, and/or invasively modifying the constituent interpreters.

Since, traditionally, interpreters are slow, composing them tends to worsen performance [3]. Fortunately, composed interpreters are as amenable to meta-tracing – and its close cousin dynamic partial evaluation [32] – as their constituent interpreters. Existing examples of such compositions include Python and Prolog [3] and Ruby, C and JavaScript [17]. Both systems have good peak performance.

In the rest of this paper, we use 'interpreter' to mean the source-code of the system that is used to produce an executable binary 'VM'. In practice, most readers can consider the terms 'interpreter' and 'VM' to be interchangeable with only a small loss of precision.

2.3 PHP

PHP is a language used widely for server-side webpage creation. Originally intended as a language to glue together CGI programs written in C, it gradually evolved into a complete programming language. Largely due to this gradual evolution, PHP features a number of design decisions that appear unusual when viewed from the perspective of other imperative languages such as Python or Ruby. As one example that appears later in the paper shows, many of PHP's primitive data types, including arrays, are immutable. We give further details on such features as needed.

PHP's syntax is Perl-esque, taking influence from the Unix shell (e.g. variables start with a '$') and C (e.g. basic control structures). The following (contrived) example shows most of the syntax needed to understand this paper's examples:

```
1 $i=1;                 // Assign 1 to variable $i
2 $j=&$i;               // Create a reference to var $i
3 $a=array(3, 4, 5);    // Create a list-like array
4 $b=array("bob"=>45);  // Create a dictionary-like array
5 foreach ($a as $c) {  // Iterate over the array
6   echo $c . " ";      // Print out (in order) 3 4 5
7 }
8 $o=new C();           // Create a new object
9 $o->m($i);            // Call method m
```

```
10 $s = <<<EOD           // Start multiline string
11   A string
12   with multiple lines
13 EOD;                  // End multiline string
```

3 PyHyp

PyHyp is a language composition of PHP and Python, implemented by composing two existing RPython interpreters: HippyVM (for PHP) and PyPy (for Python). PyPy is an industrial-strength Python interpreter which can be used as a drop-in replacement for CPython 2.7.8; HippyVM is a partially complete PHP 5.4 interpreter. PyHyp is a 'semantics preserving' composition in the sense that it adds behaviour to both Python and PHP, but does not alter or remove existing behaviour.

PyHyp programs currently start by executing PHP code. There is no deep reason for this choice, and one could easily allow it to start by executing Python code instead. Since they start as normal PHP programs, 'raw' PyHyp programs require `<?php ... ?>` tags around the entire file. In the interests of brevity, we omit these in all code listings.

We stage our explanation of PyHyp as follows: the design and implementation of its FFI (Section 4); its support for syntactic composition (Section 5); and finally cross-language variable scoping (Section 6).

4 PyHyp FFI

PyHyp defines an FFI which is the core upon which advanced functionality is later built. A simple example of PHP code using the FFI to interact with Python is as follows:

```
1 $random = import_py_mod("random");
2 $num = $random->randrange(10, 20);
3 echo $num . "\n";
```

The code first imports Python's `random` module into PHP (line 1) before calling the Python `randrange(x, y)` function to obtain a random integer between 10 and 20 (line 2) which is then printed (line 3). The only explicit use of the FFI in this example is the call to `import_py_mod`. However, the FFI is implicitly used elsewhere: the PHP integers passed as arguments to `randrange` are converted to Python integers, and the result of the function is converted from a Python integer to a PHP integer. As this may suggest, the FFI is two-way and Python code can also call PHP.

4.1 FFI Design

Many parts of PyHyp's FFI are fairly traditional, while some are unusual due to the semantic friction between Python and PHP. To the best of our knowledge there has not previously been an FFI between Python and PHP, so our solutions are necessarily novel.

4.1.1 Data Type Conversions

All FFIs have to define data type conversions between their constituents. Since primitive data types in both PHP and Python are immutable, PyHyp directly maps them from one language to the other (e.g. a PHP integer is transformed into a Python integer). Arbitrary user objects cannot be directly mapped and are instead wrapped in an *adapter* which allows the other interpreter to work transparently with the underlying foreign instance. A PHP object, for example, appears to Python code as a normal Python object, whose attributes

and methods can be accessed, introspected etc. Passing an adapter back to the language from which it was created simply removes the adapter. Adapters are immutable, only ever pointing to single object in their lifetime; the trace optimiser is then extremely effective at removing the overhead of repeated adaptations.

Collection data types are more involved. Python separates the notion of a list (i.e. resizeable array) from that of a dictionary (i.e. hashmap). In contrast, PHP has a single dictionary type called, somewhat confusingly, an array. PHP arrays are therefore also used wherever 'lists' are required. This presents an interesting case of semantic friction. Python lists and dictionaries can both be sensibly adapted in PHP as arrays. PHP arrays passed to Python, in contrast, are ambiguous: should they be adapted as lists or dictionaries? It is easy to design schemes which can be dangerously subverted. For example, a PHP array which 'looks like' a list might seem best adapted as a Python list, but later mutation to its keys (e.g. adding a string key) can turn it into something which is clearly not a list.

The only consistent design is therefore to default to adapting PHP arrays as Python dictionaries. However, users often know that a given PHP array is, and always will be, equivalent to a list. Therefore, PHP arrays adapted as Python dictionaries have an additional method `as_list`, which re-adapts the underlying array as a Python list. Whenever operations on the list adapter are called, a check is made to see whether the underlying PHP array is still list-like; if it is not (e.g. because a non-integer key has been added) an exception is raised.

In general, converting an adapted object to its 'host' language simply requires removing the adapter. The one exception is a Python list which has been passed to PHP, adapted as a PHP array, and which is subsequently returned back to Python. Since our data-conversion rules dictate that PHP arrays are adapted as Python dictionaries, Python code expecting a PHP array would get a surprising result if the PHP array returned was unwrapped directly to a Python list (rather than a dictionary). Thus a Python list adapted as a PHP array and then returned to Python has a special Python dictionary adapter. Only if `as_list` is called on that adapter is the underlying Python list returned.

4.1.2 Mutability

PHP data types are immutable except for objects (which are mutable in the same way as objects in Python) and references. Immutable data types often use copy-on-write semantics. For example, appending to an array creates a copy with an additional element at the end. Operations on references – mutable cells which typically point to immutable data – are passed onto the underlying datum, which may be replaced. For example, appending to a reference which points to an array mutates the reference to point to the newly copied array.

Since it is common to wrap PHP arrays in a PHP reference, and since Python's expectations are that such data types are mutable, PyHyp does not directly adapt arrays: instead, arrays are replaced by references to arrays, which are then adapted. Put another way, PHP arrays always appear to Python as mutable collections, whether adapted as lists or dictionaries, and those mutations are visible to PHP code as well.

4.1.3 Cross-language Calls

Both Python and PHP functions can be adapted and passed to the other language where they can be called naturally. There are, however, two cases of semantic friction: Python functions with keyword arguments; and PHP's pass-by-reference mechanism.

Simplifying slightly, Python functions accept zero or more mandatory, ordered arguments, and zero or more unordered, keyword arguments, each of which has a default value. Function calls must pass parameters for each ordered argument and then zero or more keyword arguments. The following example shows such a function and an example call:

```
1 def fmturi(host, path, scheme="http", frag="", query=""):
2   uri = "%s://%s%s" % (scheme, host, path)
3   if query: uri += "?%s" % query
4   if frag: uri += "#%s" % frag
5   return uri
6 fmturi("google.com", "/", frag="q=ecoop")
```

While PHP allows parameters to have default values, arguments must be passed in order, and there is no notion of keyword arguments. To enable PHP to call Python functions such as `fmturi`, PyHyp adds a global PHP function `call_py_func(f, a, k)` where `f` is an adapted Python function, `a` is an array of regular arguments, and `k` is an array of keyword arguments. From PHP, one can thus emulate the function call from line 6 as follows:

```
1 call_py_func($fmturi, array("google.com", "/"), array("frag" => "q=ecoop")).
```

By default, PHP parameters are pass-by-value but function signatures can mark their arguments as being pass-by-reference by prepending the parameter name with `&`. When a function with such a parameter is called, a reference is created which points to the argument passed (if it is not already a reference). Thus one can write a PHP function which swaps the contents of the variables passed to it:

```
1 function php_swap(&$x, &$y) {
2   $tmp = $y;
3   $y = $x;
4   $x = $tmp;
5 }
6 $a = 10; $b = 20;
7 php_swap($a, $b);
8 echo "$a $b\n"; // prints "20 10"
```

As this example shows, the code calling `php_swap` has no control over whether it is passing arguments with pass-by-value or pass-by-reference semantics—indeed, calling `php_swap` updates `$a` and `$b` so that they point to the newly created references. Since PyHyp needs to allow Python functions to be used as drop-in replacements for PHP functions, we need a notion of pass-by-reference for Python function arguments. This is tricky since Python has no explicit notion of a reference.

PyHyp takes a two-stage approach to reducing this case of semantic friction. First, we introduce an explicit `PHPRef` adapter into Python which represents a mutable PHP reference. `PHPRef`s support two explicit operations: `deref()` returns the value inside the reference; and `store(x)` mutates the reference to point to `x`. Second, we add a Python decorator `php_decor` which takes a keyword argument `refs` which specifies (as a sequence of argument indices) which arguments are pass-by-reference. With this, we can write a Python swap function as follows:

```
1 @php_decor(refs=(0, 1))
2 def py_swap(a, b):
3   tmp = a.deref()
4   a.store(b.deref())
5   b.store(tmp)
```

Although it may be tempting to think that `PHPRef`s should be transparent in Python, as they are in PHP, we found such a scheme to be impractical: it would require changing every possible part of the Python language and implementation where one can read or write to variables. Explaining the effects to users would be extremely challenging, as would changing

the implementation. In PyPy, for example, we estimate that this would involve changing around 100 separate locations.

Because `PHPRef`s are explicit, calling a PHP function with pass-by-reference arguments from Python is possible but, inevitably, somewhat clunky. Pass-by-reference arguments must be explicitly passed a `PHPRef` object; other object types lead to a run-time exception. Thus Python can call `php_swap` as follows:

```
xref, yref = PHPRef(x), PHPRef(y)
php_swap(xref, yref)
x, y = xref.deref(), yref.deref()
```

4.2 PyHyp FFI Internals

Until now, we have detailed the language PyHyp presents to the user. We now consider PyHyp's internal implementation details. PyHyp required modifying both HippyVM and PyPy. We added modules to both HippyVM (`pypy_bridge`) and PyPy (`hippy_bridge`), which encapsulate most of PyHyp's behaviour. Most of the common behaviour resides in the `pypy_bridge` module, though it could just as easily reside in `hippy_bridge`. Some behaviour is implemented by invasively modifying existing HippyVM or PyPy code.

4.2.1 Data Type Conversion

Viewed from a suitable level of abstraction, both HippyVM and PyPy implement their respective languages using broadly similar data type hierarchies: a root data type class – not entirely coincidentally, called `W_Root` in both interpreters – from which all objects inherit. Generally PyHyp adapters extend, directly or via a subclass, one of the `W_Root`s.

We added methods to both root data type classes: a `to_py` method to every PHP data type; and a `to_php` method to every Python data type. Calling `to_py` on a PHP datum creates a Python adapter (and vice versa for `to_php`). The default implementations of `to_py` and `to_php` return generic adapters, but other data types override them to return specialised adapters. Calling `to_py` / `to_php` on an adapter simply returns the adapted datum. The only exception is calling `to_py` on a Python list which has been adapted as a PHP array; rather than returning the Python list itself, PyHyp is forced to return a special (new) variant of a Python dictionary (see Section 4.1.1).

The generic adapters – one each in HippyVM and PyPy– simply forward attribute lookups, method calls, and the like onto the adapted object. PyHyp then defines a number of specialised adapters: 10 additional Python adapters and 8 additional PHP adapters. Some of the special adapters expose different behaviour to the user (e.g. collection data types), whereas some deal with low-level differences between data types in the VM (e.g. PyPy's layout requires separate adapters to be defined for functions and methods). As PyPy uses storage strategies to optimise collection data types [5], adapted PHP collections create Python collection instances that use PyHyp-specific strategies rather than subclassing `W_Root`.

The code for adapters is self-contained and relatively simple. Together, the PHP and Python adapters are just under 1400LoC, of which 400LoC implements new storage strategies.

4.2.2 Mutability

In order to make PHP arrays mutable from Python, PyHyp requires PHP arrays passed to Python to be wrapped in a reference. However, simply adding a reference when an adapter

is created would lead to mutations from Python not being seen by PHP. PyHyp therefore handles arrays specially: the following two examples give a flavour of this.

The `ARG_BY_PTR` PHP opcode organises function arguments prior to a function call, and is where arrays passed to a call-by-reference function argument have their storage in the PHP frame replaced by a reference. PyHyp adds a special case to this opcode: every PHP array passed to a Python function is treated as if it was being passed to a call-by-reference function argument. This ensures that the PHP frame observes mutations from Python.

Similarly, when Python loads an array from an adapted PHP object's attribute, PyHyp must replace the attribute with a reference. This ensures that the parent object observes mutations from Python. Implementing this is fairly simple, as we reuse an existing function in HippyVM which can lookup an attribute and turn non-references into references (used to implement PHP's standard `x=&$y->z` behaviour). We add a new flag to this function, since we only want to turn arrays (and not other data types) into references.

4.2.3 Cross-language Calls

For the most part, cross-language calls are simple to implement. PyHyp is careful to ensure that the necessary glue code is optimised and does not obstruct meta-tracing's natural cross-language inlining. Most importantly, this requires annotating the relevant RPython functions as being unrollable, so that they dynamically specialise themselves to the number of parameters passed by the user.

The `php_decor` decorator is implemented as a normal Python (not RPython) class. When the decorator is applied with the `refs` keyword argument, the argument indices are stored in a normal, user visible attribute of the function object. When the function is adapted for PHP, the adapter loads the indices, which are later used by the PHP interpreter to determine which parameters are to be passed by reference.

4.2.4 Transparency

Ensuring that adapters are as transparent as possible inevitably requires invasive modifications of HippyVM and PyPy. We were helped by the fact that both interpreters centralise all the behaviour we wished to modify. For example, implementing identity transparency in PyPy is easy, as identity checks are not handled directly but handed over to an object's `is_w` method which compares the current object with another for identity equality. PyHyp adapters override this method so that if the objects being compared are both adapters of the same type, the identity check is forwarded on to the underlying adapted objects. The `is_w` method of `W_PHPGenericAdapter` (the class representing an adapted PHP object in PyPy) shows this idiom:

```
1 def is_w(self, other):
2   if isinstance(other, W_PHPGenericAdapter):
3     return self.w_php_obj is other.w_php_obj
4   return False
```

5 Syntactic Composition

Traditionally, FFIs have made the implicit assumption that the source code of each language involved is kept in different files. In this section, we show how PyHyp allows PHP and Python code to be used within a single file. To avoid tedious duplication of explanation, we concentrate our explanation on embedding Python into PHP, though PyHyp also allows PHP to be embedded into Python.

5.1 Functions

At a low-level, PyHyp provides simple support for embedding Python inside PHP (and vice versa) as strings. For example, the `compile_py_func` function takes a string containing a single Python function and returns a Python function object that is adapted as a callable PHP instance. The following example embeds a Python function inside PHP and calls it to produce a random number between 0 and 10:

```
$src = <<<EOD
def randnum(n):
  import random
  return random.randrange(n)
EOD;
$randnum = compile_py_func($src);
echo $randnum(10) . "\n";
```

In this paper we mostly use `compile_py_func`, though `compile_py_func_global` can be used to compile a Python function and put it in PHP's global function namespace.

Although the `compile_*` functions are called at run-time, they are surprisingly fast. First, due to PHP and Python being simple languages to compile, both HippyVM and PyPy have efficient compiler implementations. Second, when JIT compiled, PyHyp caches the bytecode output of `compile_*` calls. Re-evaluating a function thus produces a new (cheap) function object while reusing the (expensive) bytecode object which underlies it.

5.2 Methods

PyHyp supports inserting Python methods into PHP classes via the `compile_py_meth(c, f)` function, where the Python method source `f` is compiled and inserted into the class named by the string `c`. However, this feature must be used carefully because HippyVM's implementation takes advantage of the fact that PHP's classes can be statically compiled. This means that, by the time a program has started running, normal PHP classes cannot be altered. This poses a problem for PyHyp because its syntactic embedding is currently performed at run-time. We work around this by enclosing a PHP class inside curly braces, which delays its compilation until run-time. We also require that all calls to `compile_py_meth(c, ...)` immediately follow the definition of `c` and that the class and all such calls be surrounded by curly brackets. This also affects sub-classes of `c`, whose execution must also be delayed by curly brackets (though not necessarily at the same point in the source code).

PHP supports a Java-like public/private/protected method access scheme, but Python has no concept of private or protected methods.[2] To model this, PyHyp extends the `php_decor` decorator (see Section 4.1.3) with an optional extra argument `access` which accepts a string access value (`"public"`, `"private"`, or `"protected"`). Similarly, the option `static` boolean argument allows static methods to be denoted. These arguments can be combined to specify that a Python method is e.g. protected and static.

5.3 Using Eco to Express Embeddings

For simple uses, the `compile` functions are tolerable as-is, but they tend to obfuscate embedded code, especially in multi-level embeddings (e.g. PHP inside Python inside PHP), where string escaping becomes onerous.

[2] Python attributes prefixed by two underscores have their names mangled, but are otherwise publicly accessible.

```
# Create PHP grammar referencing Python (and vice versa)
python = Grammar("python.eco")
php = Grammar("PHP+Python", "php.eco")
python.add_alternative("atom", php)
php.add_alternative("top_statement", python)
php.add_alternative("class_statement", python)
php.add_alternative("expr", python)
# Create Python expressions-only grammar
python_expr = Grammar("Python expressions", "python.eco")
python_expr.change_start("simple_stmt")
php.add_alternative("expr", python_expr)
```

Listing 1 Composing PHP and Python grammars in Eco. `Grammar(n, p)` loads a grammar named `n` from path `p`. `change_start` changes the start rule of a grammar. `g1.add_alternative(r, g2)` adds a new alternative to the rule `r` in grammar `g1` to the start rule of grammar `g2`.

In order to make PyHyp's syntactic composition more palatable, we make use of the Eco editor [11]. Eco allows users to compose grammars and to write composed programs. In essence, one takes a context-free grammar X and adds a reference from a rule $R \in X$ to another grammar Y. When entering language X into Eco, the user can switch to entering language Y by creating a language box. Language boxes delineate when one language ends and another starts, but do not introduce ambiguity or complexity into a parser. Eco parses as the user type, and thus always has access to parse trees for all languages involved in a composition. However, from the user's perspective, editing in Eco feels like a normal text editor, except when creating a language box or moving between nested language boxes (a fairly rare occurrence).

We first had to enable users to write PyHyp programs. Since Eco comes with a Python grammar, we only had to add a PHP grammar (which we adapted, with minor modifications, from Zend PHP) and a PHP lexer. We then used Eco's Python interface to express the join points between the two grammars (see Listing 1). Simplifying slightly, in PHP one can add Python language boxes wherever PHP statements or expressions are valid; and in Python one can add PHP language boxes wherever a Python expression is valid. From the users perspective, this means that when using Eco for PyHyp, they create a new `PHP+Python` file and start typing PHP. When they want to insert a Python function they insert a `Python+PHP` language box; when they want to insert a Python expression they insert a `Python expression` language box.

We then had to implement an exporter from parse trees relative to the composed `PHP+Python` composed grammar to PyHyp-compatible input. The exporter automatically inserts `compile_*` functions, follows `compile_py_meth`'s restrictions, and escapes arbitrarily deeply nested language boxes. For example, the `gen` Python language box in the `RNG` class in Figure 1 is exported as follows:

```
{
class RNG { ... }
compile_py_meth("RNG", "def gen(self, amount):\n \\
                        while amount > 0:\n \\
                          amount -= 1\n \\
                          yield self.pcg8_random()");
}
```

Figure 1's second language box (line 20) is more interesting. PyHyp has no explicit interface for embedding Python expressions. Instead, the expression is encoded as a callable PHP instance which is compiled and immediately called:

```
$l = call_py_func(compile_py_func("f = lambda: [x % 64 for x in rng.gen(25)];"));
```

Figure 2 Cross-language exception handling in PyHyp, showing that the stacktrace presents entries from within language boxes correctly. As an example, the first line of the stacktrace should be read as follows: "The 0th entry in the stacktrace relates to the exceptions.eco file, line 2, within the `h` function".

In the rest of the paper we use the term 'language box' to refer to an embedded language fragment irrespective of whether Eco is used or not.

5.3.1 Exceptions

When a native exception crosses the language boundary, PyHyp adapts it, before re-raising the exception. For example, if PHP calls Python code which raises the Python exception `ZeroDivisionError`, then the exception will appear to PHP as a generic `PyException`. As with all other adapters, an adapted exception crossing back to its native language (e.g. a `PyException` which percolates back to Python) simply has its adapter removed.

However, adapters on their own do not solve a crucial problem: cross-language stacktraces. By default, both HippyVM and PyPy can only print out their own frames in stacktraces, which makes cross-language exceptions seem to appear out of thin air. Equally frustratingly, frames which represent inner language boxes report incorrect line numbers, as each language box's frame assumes it starts at line 1.

PyHyp fixes both problems. First, we altered HippyVM and PyPy's stacktrace functions to call each other for their appropriate frames. Second, we added two arguments to PyHyp's compilation functions (e.g. `compile_py_func`) to allow Eco to pass the file and line offset the compiled text is relative to. The stacktrace routines then use this information to adjust the reported locations. The end result is that cross-language stacktraces are just as informative as mono-language stacktraces, as can be seen in Figure 2.

6 Cross-language Scoping

For syntactic composition to be useful, we believe that users must be able to reference variables across language boxes. PyHyp therefore allows both Python and PHP to reference variables in the other language, making syntactic composition significantly more powerful and usable. This raises a novel design challenge: what are sensible cross-language scoping rules? The major challenge is to deal with both language's expectations surrounding global

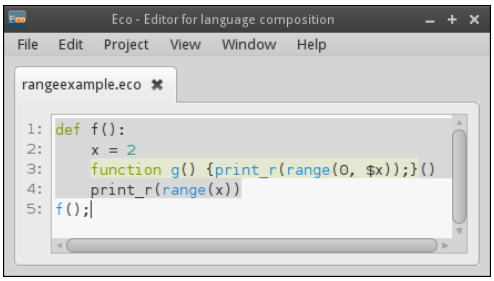

Figure 3 Cross-language scoping with nested language boxes: each language expects to see its own global scope. Lexical scoping suggests that `x` referenced on lines 3 and 4 should bind to the definition on line 2. Since PHP and Python's `range` functions are incompatible, the PHP language box on line 3 must reference a different `range` function to that on line 4. However, both the PHP and Python language boxes wish to use PHP's `print_r` function on lines 3 and 4. PyHyp's scoping rules respect these desires and this example prints out 0, 1, 2, 0, 1.

namespaces. We eventually settled upon scoping rules that are relatively easily explained, and which impose a mostly lexical system. This is no small matter, as PHP and Python have significantly different scoping rules.

In this section, we first define a simplified version of the scoping rules in each language (since PHP and Python share neither common semantics nor terminology, we do our best to homogenise the description). We then define PyHyp's additional rules for PHP referencing Python and vice versa, before explaining how these rules are implemented.

6.1 PHP and Python's Namespace Semantics

PHP has separate global namespaces for functions, classes, constants, and variables. Any given name can appear in multiple namespaces, as syntactic context uniquely identifies which namespace is being used (e.g. `new x()` references a class, whereas `x()` references a function). PHP has no concept of modules, textually 'including' files in similar fashion to C headers. Thus the global namespaces span all PHP files.[3] The namespaces for functions, classes, and constants can have new names added to them dynamically, but existing names cannot be removed or changed. In contrast, names can be added or removed to the namespace for variables at will. Each function then defines a local scope; variable lookups within a function first search the local scope before searching the global variable namespace.

Python modules each have a single 'global' namespace, to which names can be added and removed. Functions define their own local namespace. Lexical lookups (for names defined in the current or parent function's scope) are determined statically (the set of local names cannot be modified); global lookups are performed dynamically.

Neither PHP nor Python has Scheme-esque closures[4]: nested functions can read a parent function's variables but writes are not shared between the two. In the interests of brevity, we consider PHP and Python's variable scoping rules to be local and global, but not lexical.

[3] Although PHP 5.3 introduced a mechanism for defining non-global namespaces, this does not effect our explanation.
[4] For our purposes, we consider what PHP calls a 'closure' to be a first-class anonymous function.

6.2 PyHyp's Cross-language Scoping Rules

The use of cross-language scoping in Figure 1 may suggest that cross-language scoping design is a matter of applying traditional language design principles. Figure 3 shows a more challenging example, where modern expectations about lexical scoping and PHP and Python's global namespaces appear to clash.

PyHyp resolves this issue with the following simple design. First, PHP and Python's local scoping rules remain unchanged. Second, we split the search for variables which are not bound in the current language box into two distinct phases: the 'recursive' phase, and the 'global' phase. The recursive phase searches language boxes (from inner to outer) for a matching variable definition. Failing this, the global phase searches the global namespace(s) of the current language box's language; if no match is found, it then searches the global namespace(s) of the other language.

The recursive phases for PHP and Python are conceptually identical. Python's global phase is simple, but PHP's is complicated by its multiple global namespaces. If the search originated from PHP, then only the appropriate namespace for the syntactic context is used (e.g. if, syntactically, a function was looked up, only the function namespace is searched); if the search originated in Python the namespaces are searched in the following order: global functions, classes, then constants.

Performance reasons led us to make one small adjustment to the PHP global phase search. PHP has the ability to lazily load classes; every failed class lookup triggers a (fairly slow) check for user-defined lazy loading mechanisms. In mono-PHP this is a sensible mechanism, as in practice either a class is found in the namespace, or it is lazily loaded, or an error is raised. However, since cross-language scoping frequently checks for the existence of names that do not, and will never, exist, the cumulative performance effect can be frustrating. Since disabling lazy loading would break many existing PHP applications, we tweaked PyHyp's scoping rules. The search for a Python name in PHP's global namespaces is 'sticky': if a name x was found to be (say) a class on the first search in a given scope, it will only ever return a class on subsequent searches (i.e. if a function x is later added to the functions namespaces, it will not be returned as a match in that scope). This small loss of dynamicity increases performance in some benchmarks by around 50%.

Using Figure 3 as a concrete example, we can see how these rules apply in practice. First, consider the PHP variable reference `$x` on line 3. There is no binding of `x` in the PHP language box so when the code is executed a recursive phase search commences: the parent (Python) language box is inspected and a binding found on line 2. The `range` function reference on line 3 starts with the same pattern: a recursive phase search looks in the Python language box for a binding but fails. A PHP global phase search then commences; since `range` was syntactically referenced as a function, only the global function namespace is searched, where a match is then found. The `print_r` function reference on line 3 follows the same pattern. The reference to a `range` function in the Python language box at line 4 starts with a recursive phase search which looks into the parent PHP language box's for a suitable binding (i.e. a name starting with a '`$`') but fails. It then does a global phase search in Python's global namespace, finding Python's built-in `range` function. The `print_r` reference on line 4 is more interesting. A recursive phase search fails to find a match. A global phase search then searches in Python's global namespace and fails before trying PHP's global namespaces, starting with functions, where a match is found.

PyHyp's scoping rules also work well in corner-cases (e.g. PyHyp deals with PHP's superglobals sensibly). Note that the scoping rules of both languages are partly or wholly dynamic: that is, in some situations, bindings can be changed at run-time. PyHyp's scoping

rules maintain PHP's and Python's dynamic lookup properties since some programming idioms (particularly in PHP, but also in Python) rely on adding or removing bindings.

6.2.1 Implementation

To add PyHyp's scoping rules to HippyVM and PyPy, we first needed to connect language box scopes together at run-time, and then intercept the locations where PHP and Python global variables are read and written to.

Connecting language box scopes is made relatively simple by the fact that each is constructed by a `compile_*` function (see Section 5.1). The outer part of any PyHyp program is, by definition, a PHP language box and every other language box is nested inside that. Thus any call to `compile_py_*` implicitly receives a reference to the PHP frame from which it was called. The reference is then stored in a `PHPScope` object, which PyHyp attaches to the Python function object being created; nested Python functions inherit a `PHPScope` object from their parent function, so that multiply nested functions can still access outer language boxes. Similarly, when a PHP language box is nested inside Python, a `PyScope` object is created and placed inside a PHP function's bytecode object. This simple scheme means that, from any PHP or Python frame, one can walk a chain from the current to the outermost language box.

To actually search outer language box's scopes, we have to modify those parts of HippyVM and PyPy which perform global lookups. In HippyVM, we modify the 3 separate functions on the main interpreter object which perform searches of functions, classes, and constants, as well as the `lookup_deref` function on frames which lookups up variables. An elided version of the `locate_function` function – which searches for a PHP function `n` – shows the small scale of such modifications:

```
1  def locate_function(n):
2    py_scope = self.topframeref().bytecode.py_scope
3    if py_scope is not None:
4      ph_v = py_scope.ph_lookup_local_recurse(name)
5      if ph_v is not None: return ph_v
6      ph_v = self.lookup_function(name)
7      if ph_v is not None: return ph_v
8      ph_v = py_scope.ph_lookup_global(name)
9      if ph_v is not None: return ph_v
10   else:
11     func = self.lookup_function(name)
12     if func is not None: return func
13   self.fatal("Call to undefined function %s()" % name)
```

In essence, the original function consisted of lines 11 and 12; PyHyp adds lines 1–9. When a PHP function performs a global function lookup, and that PHP function is nested inside a Python language box (lines 1 and 2) then a local phase search is performed (lines 4 and 5). If unsuccessful, the global phase search then commences: first the PHP global function namespace is searched (lines 6 and 7) before Python's global namespace is searched (lines 8 and 9). The modifications made to PyPy are identical in idiom, modifying the two opcodes (`LOAD_GLOBAL` and `STORE_GLOBAL`) which read and write global variables.

Since PyHyp's rules are highly dynamic, we rely heavily on use of Self-style maps (see [9]) to turn most name lookups from (slow) dictionary lookups of names into (fast) constant lookups. HippyVM originally made only limited use of maps in its global namespaces: we altered it to use maps extensively.[5] We also used maps for the sticky namespace search. Looking up a variable in a global phase search in a PHP scope returns an integer representing

[5] This optimisation also helps plain HippyVM, as it significantly improves the performance of programs such as MediaWiki and phpBB that use the `$GLOBALS` superglobal.

unknown (i.e. this name has not previously been searched for in this context), class, function, constant, or not found (i.e. a search was previously done and no match was found). After tracing, virtually all global lookups turn into a small number of (quick) attribute guards, avoiding (slow) dictionary lookups.

7 Experiment

To understand PyHyp's performance characteristics, we define three classes of benchmarks: small microbenchmarks, large microbenchmarks, and permutation benchmarks. Benchmarks come in four variants: mono-language PHP (henceforth 'mono-PHP'); mono-language Python ('mono-Python'); composed PHP and Python where PHP is the 'outer' language ('composed-PHP') and where Python is the 'outer' language ('composed-Python'). We run these benchmarks on several PHP and Python implementations.

7.1 Benchmarks

Our small microbenchmarks, focus on single language features in isolation, and are useful for identifying low-level pinch points. Each of our small microbenchmarks consist of two parts. In most, an outer loop repeatedly calls an inner function (e.g. the *total_list* benchmark's inner function takes a list of integers and sums them). In the remainder, an outer function generates elements, and an inner function consumes them. In the composed variants, the inner and outer components are implemented in different languages.

Some small microbenchmarks cannot be implemented in all variants. The *ref_swap* benchmark measures the performance of operations on PHP references and `PHPRefs` (see Section 4.1.3) and thus has no mono-Python variant. Benchmarks which require putting PHP methods into Python classes are currently not supported by PyHyp. The complete list of small microbenchmarks can be found in Appendix A.

Our large microbenchmarks aim to measure performance more broadly. We use four 'classic' benchmarks: *Fannkuch* counts permutations by continually flipping elements in a list [1]; *Mandel* plots an ASCII representation of the Mandelbrot set into a string buffer[6]; *Richards* models an operating system task dispatcher[7]; and *DeltaBlue* is a constraint solver [27]. To create composed variants of these benchmarks, we took the mono-language variants and replaced each function with an implementation in the other language. The composed-PHP variants of Richards and DeltaBlue are thus PHP 'shell' classes containing many Python methods (33 and 75 respectively), with variables referenced between languages and data repeatedly crossing from PHP to Python (Figure 4 shows the mono and composed-PHP variants of Richards alongside each other). In other words, Richards and DeltaBlue are designed to heavily test PyHyp's cross-language performance. In contrast, Fannkuch is a single function, and so the composed variant consists of a single Python function embedded in PHP. This serves as a rough baseline, since we would expect that the composed variant has roughly the same performance as PyPy. Mandel also started off life as a single function, but we made modifications to make a more interesting benchmark: we split the innermost loop into a separate function; made the function's parameters pass-by-reference; and made the function modify these references during execution. Since Mandel uses references, there

[6] `Zend/bench.php` in the Zend distribution of PHP.
[7] `http://www.cl.cam.ac.uk/~mr10/Bench.html`

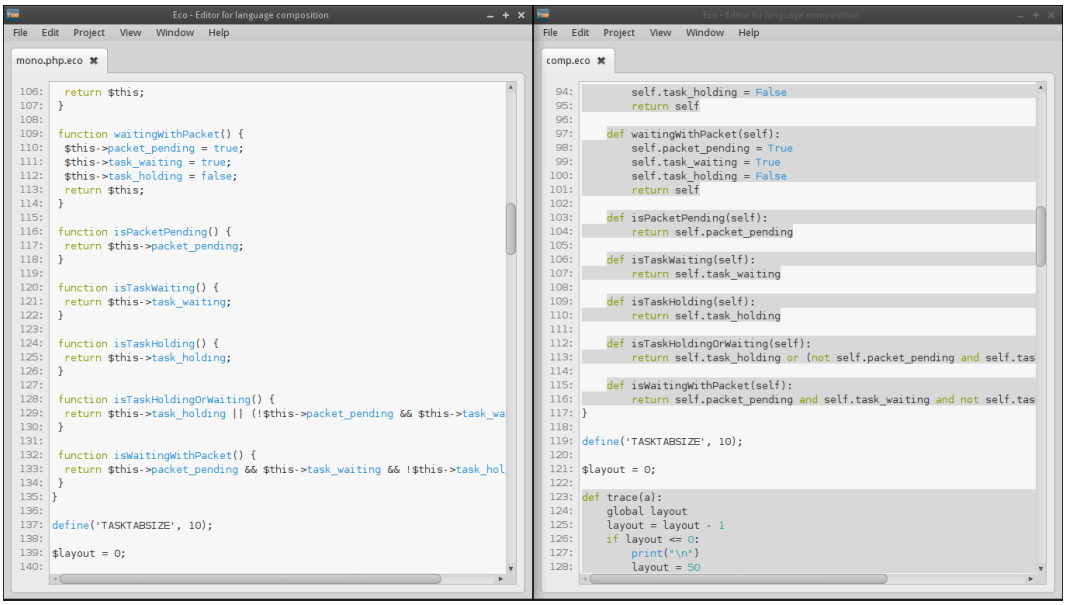

Figure 4 The mono-PHP and composed-PHP variants of Richards side by side. The composed-PHP variant of the benchmark contains a 'shell' PHP program with PHP classes whose methods are Python language boxes. Global variables remain defined in PHP, so that the benchmarks also include an element of cross-language scoping.

is no mono-Python variant. There is no composed-Python variant of either Richards or DeltaBlue, since PHP methods cannot yet appear inside Python classes.

The permutation benchmarks are designed to uncover whether some parts of a program are faster in one or other language. Using the mono-PHP DeltaBlue benchmark as a base, we created 79 permutations, each with one PHP function replaced by a Python equivalent. We then compare the timings of each permutation to the original mono-PHP benchmark. For brevity, we henceforth refer to permutation number x as p_x.

7.2 Methodology

Each benchmark was run on the following VMs (in alphabetical order): CPython, the standard interpreter for Python; HHVM, a JIT compiling VM for PHP; HippyVM; PyHyp$_{\text{PHP}}$, which is PyHyp running composed-PHP variants; PyHyp$_{\text{Py}}$, which is PyHyp running composed-Python variants; PyHyp$_{\text{mono}}$, which is PyHyp running mono-PHP variants; PyPy; and Zend, the standard interpreter for PHP. The versions used for each of these VMs is shown in Table 1. Note that PyHyp$_{\text{PHP}}$, PyHyp$_{\text{Py}}$ and PyHyp$_{\text{mono}}$ are all the save VM, and we use the terms to be clear about which benchmarks PyHyp is running.

For each benchmark and VM pair, we ran 5 fresh processes, with each process running 50 iterations of the benchmark. We then used the bootstrapping technique described in [21] to derive means and 99% confidence intervals for each pairing. Since we could not always determine when a given VM had warmed up, we made no attempt to remove any iterations from the process: thus our timings include warmup. All timings are wall-clock with a sub-microsecond resolution.

All experiments were run on an otherwise idle 4GHz Core i7-4790 CPU and 32GiB RAM machine, running Debian 8. We disabled turbo mode and hyper-threading in the BIOS. We

Table 1 The VM versions used in this paper.

Interpreter	Version(s)
CPython	2.7.10
HHVM	3.4.0
HippyVM	git #2ae35b80
PyPy	2.6.0
Zend	5.5.13
PyHyp	Based on above HippyVM/PyPy versions.

used a tickless Linux kernel, disabled Intel p-states, and ensured that the CPU governor was set to maximum performance mode. All VMs were built with the system GCC (4.9.2). We did not interfere with the garbage collection of any of the VMs, which run as normal.

7.3 Results

Table 2 shows the results of our microbenchmarks relative to the composed PyHyp variant. Absolute timings are shown in Table 4 in the Appendix. Starting with the simplest observations, we can see that Zend and CPython (C-based interpreters) are slower than HHVM, HippyVM, and PyPy (JIT-based VMs). Small to medium benchmarks tend to flatter meta-tracing, and so HippyVM and PyPy outperform HHVM. PyPy is nearly always faster than HippyVM, reflecting the greater level of engineering PyPy has received.

$\text{PyHyp}_{\text{mono}}$'s results are very similar to HippyVM's, though a few cases run slightly slower on PyHyp. For walk_list and DeltaBlue, some missing optimisations in PyHyp's scoping lookups cause undue bloat in the optimised traces. instchain and sum_meth_attr in contrast have identical traces except for a small portion of their headers: this seemingly small difference has a surprisingly large run-time effect which we do not fully understand.

PyHyp is generally faster than HippyVM on the composed-PHP benchmarks. This is largely due to moving code from PHP (slower HippyVM) to Python (faster PyPy) and the ability that meta-tracing has to naturally inline code across both languages. PyHyp is in most cases slower than PyPy, as we would expect, because of the additional overhead of adapters and cross-language scoping. Although meta-tracing naturally optimises the vast majority of these operations away, a few inevitably remain, and their cumulative effect is often noticeable, even though it is not severe. On the geometric mean $\text{PyHyp}_{\text{PHP}}$ is only around 20% slower on average than PyPy, and no individual $\text{PyHyp}_{\text{PHP}}$ benchmark is more than 2.2x slower. The composed-Python benchmarks have a similar overall average to the composed-PHP benchmarks, though several benchmarks are slower. By comparing traces from the composed-PHP and composed-Python benchmarks, we were able to identify several missing optimisations in HippyVM that are likely to account for most such slowdowns: redundant comparisons in logical operators; many more allocations in PHP iterators; and more allocations when appending to PHP lists.

In some cases, the timings for composed variants running on PyHyp are virtually identical to the mono-language variants running on PyHyp's constituents (e.g. smallfunc on HippyVM, $\text{PyHyp}_{\text{PHP}}$ and PyPy are all roughly 1x). For small benchmarks, we would expect any well-written RPython VM to compile virtually identical traces, and such bench-

Table 2 Microbenchmark timings relative to $\text{PyHyp}_{\text{PHP}}$. Note that $\text{PyHyp}_{\text{PHP}}$ and PyHyp_{Py} are the same VM, but running composed-PHP and composed-Python benchmark variants respectively.

Benchmark	CPython	HHVM	HippyVM	$\text{PyHyp}_{\text{PHP}}$	PyHyp_{Py}	$\text{PyHyp}_{\text{mono}}$	PyPy	Zend
instchain	33.451 ±0.0679	9.547 ±0.0096	0.912 ±0.0011	1.000		1.116 ±0.0012	0.675 ±0.0007	36.471 ±0.1577
l1a0r	86.017 ±0.0179	4.052 ±0.0020	1.368 ±0.0004	1.000	1.360 ±0.0003	1.359 ±0.0003	1.340 ±0.0106	38.778 ±0.0078
l1a1r	83.803 ±0.1407	2.980 ±0.0038	1.306 ±0.0017	1.000	1.303 ±0.0016	1.303 ±0.0016	1.140 ±0.0022	39.111 ±0.1272
lists	8.047 ±0.0139	0.931 ±0.0036	0.975 ±0.0020	1.000	0.560 ±0.0012	0.978 ±0.0021	0.497 ±0.0010	14.626 ±0.0377
ref_swap		8.393 ±0.0006	1.000 ±0.0002	1.000	0.700 ±0.0001	1.000 ±0.0001		53.320 ±0.0040
return_simple	110.409 ±0.1104	7.049 ±0.0019	1.000 ±0.0001	1.000	0.778 ±0.0001	1.000 ±0.0001	0.889 ±0.0001	84.724 ±0.0645
scopes	133.487 ±0.0493	15.023 ±0.0018	4.511 ±0.0025	1.000	0.929 ±0.0005	4.495 ±0.0013	1.000 ±0.0001	152.608 ±0.0131
smallfunc	187.132 ±0.1488	13.078 ±0.0010	1.000 ±0.0001	1.000	0.750 ±0.0000	1.000 ±0.0001	1.000 ±0.0001	230.818 ±0.0145
sum	317.479 ±0.2718	19.362 ±0.0014	0.999 ±0.0001	1.000	0.750 ±0.0001	1.000 ±0.0003	0.874 ±0.0001	418.485 ±0.0921
sum_meth	341.850 ±1.3274	24.106 ±0.0280	0.999 ±0.0001	1.000		1.000 ±0.0001	0.874 ±0.0002	447.472 ±0.4913
sum_meth_attr	131.469 ±0.7392	17.915 ±0.1052	0.999 ±0.0061	1.000		1.131 ±0.0065	0.904 ±0.0057	145.365 ±0.8321
total_list	19.230 ±0.0145	2.245 ±0.0008	0.864 ±0.0002	1.000	1.508 ±0.0004	0.858 ±0.0005	0.587 ±0.0003	33.667 ±0.0633
walk_list	5.060 ±0.0071	0.406 ±0.0005	0.779 ±0.0011	1.000	1.601 ±0.0026	1.010 ±0.0018	1.080 ±0.0015	10.647 ±0.0677
deltablue	16.528 ±0.0707	671.482 ±2.9041	4.325 ±0.0212	1.000		4.507 ±0.0214	0.457 ±0.0026	144.149 ±2.6843
fannkuch	20.582 ±0.0226	3.342 ±0.0025	1.848 ±0.0007	1.000	1.891 ±0.0005	1.878 ±0.0005	1.005 ±0.0004	14.387 ±0.0128
mandel		0.791 ±0.0056	0.921 ±0.0005	1.000	0.493 ±0.0001	0.999 ±0.0003		7.241 ±0.0188
richards	26.902 ±0.0189	11.897 ±0.0088	0.853 ±0.0010	1.000		0.887 ±0.0007	0.488 ±0.0005	24.207 ±0.0236
Geometric Mean	52.743 ±0.0341	6.940 ±0.0047	1.222 ±0.0006	1.000	0.963 ±0.0003	1.277 ±0.0006	0.813 ±0.0007	55.549 ±0.0692

marks show this effect. However, in some cases where we expected identical performance for both PyHyp and its constituents, the composed variant is faster: lla0r, l1a1r, and smallfunc. Indeed, the crucial parts of the traces were identical between the two VMs in all these cases. Further exploration showed that RPython's machine code generator occasionally emits less than optimal code (in this case unnecessary x86-64 `MOV`s) that account for the difference. We do not understand the precise reason for this, but it seems plausible it is a limitation of the current register allocator. We have reported our findings to the RPython developers.

Table 3 shows the results from the permutations experiment. The majority of permutations are statistically indistinguishable from mono-PHP; most of the remainder are close enough in performance to be of little interest. Four permutations, however, show substantial differences: p_2, p_5, p_6 and p_7 all perform much better than in mono-PHP. We now describe the reasons for these cases.

p_2 swaps the `OrderedCollection` class's constructor which performs a single action, assigning an array (in PHP) or list (in Python) to the `elms` attribute. The seemingly innocuous change of moving from a PHP array to a Python list has a big impact on performance simply because PyPy's lists are are far more extensively optimised than HippyVM's (see [5]). This provides indirect evidence of the importance of making adapters immutable (see Section 4.1.1): even though p_2 operates extensively on adapters, their costs after trace optimisation are extremely small.

p_5, p_6, and p_7 are all similar in nature. Ultimately, and perhaps surprisingly, the slowdown is due to Hippy using a tracing garbage collector. Because of PHP's copy-on-write semantics, arrays are conceptually copied on every mutation. Zend (the traditional PHP implementation) is able to optimise away many of these writes by making use of its reference counting garbage collector. When a mutation operation occurs on an array with a reference

Table 3 DeltaBlue permutations in PyHyp, with absolute times (in seconds) and relative timings (to mono-PHP DeltaBlue run on PyHyp). Greyed-out cells indicate that the confidence intervals overlap. Bold entries indicate that there is more than a 25% relative performance difference.

P1: 0.246s ±0.0005 1.000× ±0.0029	P17: 0.246s ±0.0004 1.000× ±0.0028	P33: 0.246s ±0.0005 0.999× ±0.0028	P49: 0.246s ±0.0005 1.002× ±0.0028	P65: 0.246s ±0.0005 1.001× ±0.0029
P2: 0.120s ±0.0003 0.490× ±0.0015	**P18: 0.250s ±0.0005 1.015× ±0.0029**	P34: 0.246s ±0.0004 1.000× ±0.0026	P50: 0.246s ±0.0006 1.001× ±0.0032	**P66: 0.247s ±0.0004 1.006× ±0.0026**
P3: 0.240s ±0.0004 0.978× ±0.0026	**P19: 0.245s ±0.0004 0.999× ±0.0026**	P35: 0.246s ±0.0006 1.000× ±0.0031	P51: 0.246s ±0.0004 1.000× ±0.0027	P67: 0.246s ±0.0005 1.001× ±0.0027
P4: 0.249s ±0.0005 1.015× ±0.0030	**P20: 0.245s ±0.0004 0.998× ±0.0026**	P36: 0.246s ±0.0005 1.002× ±0.0030	P52: 0.246s ±0.0004 1.000× ±0.0026	P68: 0.246s ±0.0005 1.000× ±0.0030
P5: 0.132s ±0.0002 0.538× ±0.0014	P21: 0.246s ±0.0004 1.001× ±0.0027	P37: 0.246s ±0.0005 1.001× ±0.0028	**P53: 0.248s ±0.0004 1.008× ±0.0027**	**P69: 0.251s ±0.0004 1.021× ±0.0028**
P6: 0.131s ±0.0002 0.533× ±0.0013	P22: 0.247s ±0.0005 1.005× ±0.0029	P38: 0.246s ±0.0005 1.000× ±0.0028	P54: 0.246s ±0.0005 0.999× ±0.0028	**P70: 0.248s ±0.0005 1.008× ±0.0028**
P7: 0.175s ±0.0003 0.710× ±0.0020	**P23: 0.244s ±0.0005 0.993× ±0.0027**	P39: 0.246s ±0.0005 0.999× ±0.0027	P55: 0.247s ±0.0005 1.004× ±0.0028	**P71: 0.242s ±0.0004 0.986× ±0.0025**
P8: 0.246s ±0.0006 1.002× ±0.0032	P24: 0.246s ±0.0005 1.000× ±0.0029	P40: 0.246s ±0.0005 1.000× ±0.0030	P56: 0.247s ±0.0006 1.005× ±0.0031	P72: 0.246s ±0.0005 1.001× ±0.0028
P9: 0.246s ±0.0005 1.000× ±0.0029	P25: 0.246s ±0.0005 1.002× ±0.0029	P41: 0.245s ±0.0005 0.995× ±0.0028	P57: 0.245s ±0.0005 0.999× ±0.0028	P73: 0.246s ±0.0005 1.000× ±0.0027
P10: 0.246s ±0.0004 1.000× ±0.0026	P26: 0.246s ±0.0004 1.001× ±0.0027	**P42: 0.248s ±0.0004 1.011× ±0.0027**	**P58: 0.245s ±0.0004 0.999× ±0.0026**	**P74: 0.248s ±0.0004 1.011× ±0.0027**
P11: 0.246s ±0.0005 1.000× ±0.0027	P27: 0.247s ±0.0006 1.006× ±0.0031	P43: 0.247s ±0.0005 1.005× ±0.0029	**P59: 0.248s ±0.0006 1.011× ±0.0032**	**P75: 0.251s ±0.0004 1.021× ±0.0027**
P12: 0.246s ±0.0004 1.001× ±0.0027	P28: 0.247s ±0.0005 1.004× ±0.0029	**P44: 0.248s ±0.0005 1.011× ±0.0031**	P60: 0.247s ±0.0005 1.005× ±0.0030	P76: 0.246s ±0.0004 1.002× ±0.0025
P13: 0.246s ±0.0005 1.000× ±0.0029	P29: 0.246s ±0.0006 1.000× ±0.0032	**P45: 0.248s ±0.0005 1.011× ±0.0027**	**P61: 0.248s ±0.0005 1.009× ±0.0028**	**P77: 0.244s ±0.0005 0.992× ±0.0027**
P14: 0.248s ±0.0005 1.009× ±0.0029	P30: 0.246s ±0.0006 1.000× ±0.0031	**P46: 0.270s ±0.0005 1.100× ±0.0029**	P62: 0.246s ±0.0005 1.000× ±0.0027	P78: 0.246s ±0.0003 0.999× ±0.0025
P15: 0.246s ±0.0004 0.999× ±0.0026	P31: 0.246s ±0.0004 1.000× ±0.0027	**P47: 0.267s ±0.0004 1.085× ±0.0027**	**P63: 0.245s ±0.0004 0.999× ±0.0026**	P79: 0.246s ±0.0004 0.999× ±0.0026
P16: 0.246s ±0.0004 1.000× ±0.0026	P32: 0.246s ±0.0005 1.000× ±0.0028	P48: 0.247s ±0.0005 1.003× ±0.0030	P64: 0.245s ±0.0005 0.998× ±0.0028	

count of 1, the array is mutated in place, as the change cannot be observed elsewhere. HippyVM in contrast does not use reference counting, and does not know exactly how many pointers to an array exist at any given point. Checking every pointer in the run-time system would be prohibitively expensive, so HippyVM approximates Zend's optimisation with a 'unique' flag on array references. Various operations can remove uniqueness, but arrays in unique references can be optimised in the same manner as Zend arrays with a reference count of 1. Taking p_5 as a concrete example, we can see the subtle effects of this optimisation in HippyVM. p_5 swaps the `OrderedCollection` class's `size` method, which simply calls `count` (in PHP) or `len` (in Python) on a PHP array stored in an attribute. Since the `count` function is call-by-value, HippyVM optimises the copy that of the array that should occur by simply dropping its unique flag; later mutations thus must therefore copy the array. However, this is not directly why PyHyp is faster than HippyVM in p_5. When the `OrderedCollection` class's `size` method is moved to Python, PyHyp's mutability semantics (see Section 4.1.2) cause `size` to be a pass-by-reference function, thus meaning that the PHP reference does not lose uniqueness, and less copying is then required.

7.4 Threats to Validity

Benchmarks are only ever a snapshot of certain performance characteristics of a system, and we do not pretend that they necessarily tell us about program performance in a more general setting. Our experiments also make no attempt to account for JIT warmup (for reasons explained in Section 7.2). Removing JIT warmup would thus 'improve' the perceived timings of VMs such as PyHyp which perform JIT compilation. Since it is also known that RPython VMs have relatively poor warmup [6], the likely effect of our decision is to make PyHyp look worse relative to other VMs. We consider this a better trade-off than trying to make other VMs look worse relative to PyHyp.

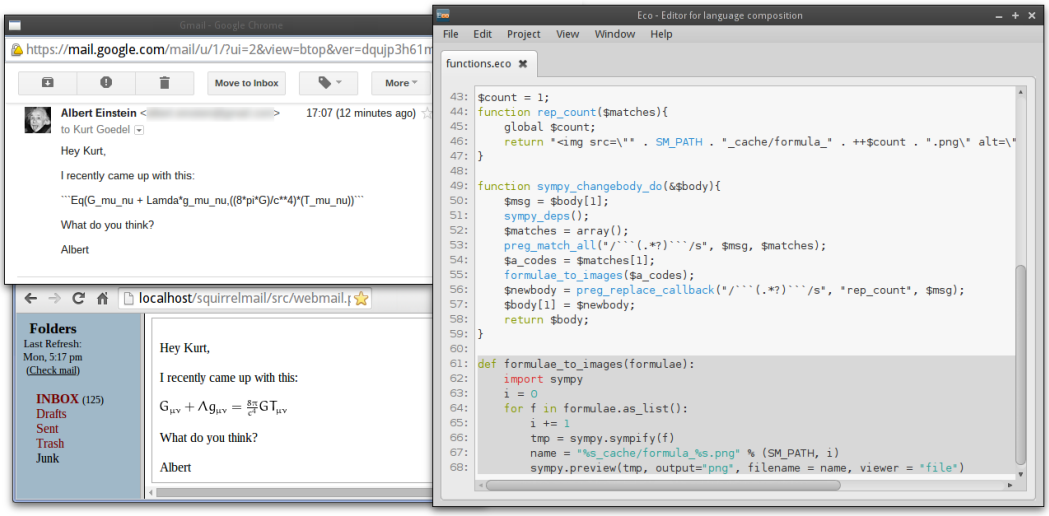

Figure 5 Example mails sent with our extended version of SquirrelMail. We extended this PHP mail client such that it can visualise mathematical formulae using the SymPy Python library. A portion of the plug-in code is shown in the right.

8 Case Studies

8.1 Using CFFI in PHP

PHP does not have a built-in C FFI, whereas Python does via the `cffi` module. PHP code can thus use PyHyp to access `cffi`, acquiring a C FFI by default. For example the following elided example shows PHP using `cffi` to call the Unix `clock_gettime` function:

```
1  $cffi = import_py_mod("cffi");
2  $ffi = new $cffi->FFI();
3  $ffi->cdef("double _clock_gettime_monotonic();");
4  $csrc = <<<EOD
5    double _clock_gettime_monotonic(){
6      struct timespec ts;
7      if ((clock_gettime(CLOCK_MONOTONIC, &ts)) == -1)
8        err(1, "clock_gettime error");
9      return ts.tv_sec + ts.tv_nsec * pow(10, -9);
10   }
11 EOD;
12 $ffi->set_source("_example", $csrc);
13 $C = $ffi->dlopen(null);
14 echo "Monotonic time: " . $C->_clock_gettime_monotonic() . "\n";
```

8.2 A SquirrelMail Plugin

SquirrelMail is a venerable PHP web mail client. We used PyHyp to add a SquirrelMail plug-in that uses the Python SymPy library. This is intended to show that PyHyp can be used to add Python modules to relatively large existing systems. In essence, the plug-in recognises mathematical formulae between triple backticks, and uses SymPy to render them in traditional mathematical notation. Formulae in incoming emails are automatically rendered; users sending emails with such formulae can preview the rendering before sending. Figure 5 shows the plug-in in use, and the core parts of the code within Eco.

The `sympy_changebody_do` function is called by SquirrelMail's `message_body` hook (which is also called upon viewing a message), receiving the content of the email as an argument. A regular expression finds all occurrences of formulae between backticks (line 53)

and passes them to the Python `formulae_to_images` function. This then uses SymPy to convert the formulae to images (numbered by their offset in the array/list) into the directory pointed to by the PHP constant `SM_PATH` (lines 61–68), and uses the URL of the image in-place of the textual formula (line 56).

8.3 System Language Migration

We expect that one of the key uses of syntactic language composition is system language migration, where systems are slowly migrated from language A to B in small stages. Instead of having to rewrite whole modules or sub-systems, syntactic language composition offers the possibility of migrating one function at a time. A full case study is far beyond the scope of this paper, but we implicitly modelled this technique when creating the DeltaBlue and Richards benchmarks, where we translated each PHP method into Python, leaving only 'shell' PHP classes, global functions and variables. As Section 7.3 clearly shows, the resulting performance is at worst 2x of its mono-language variant, which we believe makes system language migration plausible for the first time.

9 Discussion

To give an approximate idea of PyHyp's size, some rough metrics are useful. The `pypy_bridge` module – in which the majority of PyHyp is implemented – adds around 2KLoC. Aside from this we added around 0.25KLoC and 0.2KLoC to the existing HippyVM and PyPy interpreter code respectively. 5KLoC of new unit tests were added. On a fast build machine (4GHz Core i7) PyHyp takes about 45 minutes to build. We estimate that implementing PyHyp took around 7 person months.

A succinct summary of our experiences of creating PyHyp is: implementing what we wanted was fairly easy; making what we implemented run fast was somewhat easy; but working out what we wanted to implement was often hard. The latter point may surprise readers as much as it has surprised us. There are two main reasons for this.

First, there is little precedent for fine-grained syntactic language composition. Most existing language compositions are either extremely crude (per process compositions) or have design decisions implicitly imposed upon them (translating into another VM's bytecode). We therefore faced a number of novel language design issues, and used gradually larger case studies to help us iterate our way to good solutions, sometimes exhausting what felt like every possible alternative. Cross-language scoping is a good example of this: we tried many possibilities before settling on the scheme described in Section 6.

Second, it is difficult, and probably impossible, for any single person to be truly expert in every language and implementation involved in a composition. We sometimes had to base initial designs on (hopefully) intelligent guesses about one or the other languages' semantics or implementations. We then tried as hard as we could to break the resulting design. It rapidly became clear to us that in large languages such as PHP and Python, there are many corner-cases, sometimes little used, which need to be considered. PHP references caused us more headaches than any other language feature. At first we ignored them, and then we failed to appreciate their pervasive nature. It took us considerable effort to understand them well enough to make sense of their place within PyHyp. While we do not pretend to be experts in every aspect of either PHP or Python, we can recommend this route to anyone who wishes to understand the nooks and crannies of a language and its implementation.

Once we had settled upon a good design, we rarely had substantial difficulty in modifying HippyVM or PyPy to implement it. The relatively small size of the additional / changed code

in the composition is a reasonable proxy for this. Similarly, the very nature of meta-tracing meant that most cross-language optimisations came without any extra work on our part. Cross-language variable scoping was the only feature that required substantial optimisation effort on our part, including to HippyVM itself.

9.1 Generalising from the Case Study

PyHyp is, to the best of our knowledge, the first fine-grained language composition. Although we are cautious about over-generalising our results, we believe that some of the lessons embodied in PyHyp may be relevant for future fine-grained language compositions.

Most obviously, despite the rather different run-time properties of PHP and Python, PyHyp's performance is close enough to HippyVM and PyPy to be usable. While we would like to claim credit for all of this, most of the benefit comes from meta-tracing: only in a few places did we have to add PyHyp-specific optimisations. We expect languages even more disparate than PHP and Python to still achieve fairly good performance using meta-tracing.

Our use of adapters meant that most interactions between PHP and Python required little or no effort on our part to compose together satisfactorily. We expect this to generalise to most other compositions. Adapters were also the key to resolving seemingly major semantic data-type incompatibilities between (mostly immutable) PHP and (mostly mutable) Python. The techniques we used are likely to be relevant to compositions involving languages that are more rigorously immutable than PHP.

Finally, despite the archaic nature of PHP and Python's scoping rules, we were able to design good cross-language scoping rules. Most modern languages have embraced lexical scoping, and compositions involving them will require less contortions than PyHyp.

10 Related Work

There has been a long-standing desire for language composition (see e.g. [10]), and many flavours have developed since then. Extensible languages (e.g. [19, 20, 8]) aim to grow a language as required by a user. However, the base language places restrictions on what extensions are possible (e.g. due to parsing restrictions) and performant [31]. Translating one language into another (with e.g. Stratego [7]) removes many of the limitations on what is expressible, but full-scale translations are complex (e.g. [15]) and typically suffer the same performance issues as extensible languages. However, for small use cases, or where performance is not important, either approach can work well.

FFIs are the most common approach to composing languages, but their performance is typically poor due to their inability to inline cross-language calls. The next most common mechanism is to target an existing high performance VM (typically HotSpot). However, since such VMs can only optimise those programs they expect to commonly see. Languages which step even slightly outside this mould perform poorly. For example, Java programs have often excellent performance on HotSpot, but Python programs on HotSpot generally run slower than with simple C-based interpreters [28, 6].

Our aim in this paper has been to show that fine-grained syntactic language composition is possible and performant. We make no claims about the formal properties of the resulting composition as the practical challenges identified in this paper are already substantial. There is already a small body of work on formalising language composition, such as an investigation of the COM architecture [30], and an abstract framework for specifying the operational semantics of multi-language embeddings [23]. There are also partial formal semantics for

languages such as Python and PHP (see e.g. [26, 12]). We welcome future work formalising fine-grained compositions.

As the case studies show, PyHyp is at least somewhat usable, but we are under no illusions that it is an industrial strength product. There are many interesting directions for further exploration, such as experimenting with cross-language inheritance [14].

11 Conclusions

In this paper we introduced PyHyp, a fine-grained syntactic composition of PHP and Python implemented by combining together meta-tracing interpreters. We consider that PyHyp validates our hypothesis that programming languages can be composed in a finer-grained manner than previously thought possible or practical. Not only does PyHyp introduce novel concepts such as cross-language variable scoping, but its performance is close enough to its mono-language cousins to encourage use of such a system. Inevitably, some of PyHyp's details are specific to the particular pair of languages it composes. However, many of the techniques that PyHyp embodies – the use of interpreter composition with meta-tracing; some of the design choices surrounding cross-language scoping – are likely to be of use to future language compositions.

Acknowledgements We thank Armin Rigo for adjusting RPython to cope with some of PyHyp's demands, and advice on Hippy; Ronan Lamy and Maciej Fijałkowski for help with Hippy; Jasper Schulz for help with cross-language exceptions; Alan Mycroft for insightful thoughts on language composition; and Martin Berger, Darya Kurilova, and Sarah Mount for comments.

References

1 Kenneth R. Anderson and Duane Rettig. Performing Lisp analysis of the Fannkuch benchmark. *Lisp Pointers*, 7(4):2–12, Oct 1994.

2 Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI*, pages 1–12, Jun 2000.

3 Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. Approaches to interpreter composition. *COMLAN*, 44(C), March 2015.

4 Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A trace-based JIT compiler for CIL. In *OOPSLA*, pages 708–725, Mar 2010.

5 Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage strategies for collections in dynamically typed languages. In *OOPSLA*, pages 167–182, Oct 2013.

6 Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 98, Part 3:408–421, Feb 2015.

7 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *SCICO*, 72(1–2):52 – 70, 2008.

8 Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible grammars for language specialization. In *Database Programming Languages*, pages 11–31, Aug 1993.

9 Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, Sep 1989.

10 Thomas E. Cheatham. Motivation for extensible languages. *SIGPLAN*, 4(8):45–49, Aug 1969.

11 Lukas Diekmann and Laurence Tratt. Eco: A language composition editor. In *SLE*, pages 82–101, Sep 2014.

12 Daniele Filaretti and Sergio Maffeis. An executable formal semantics of PHP. In *ECOOP*, pages 567–592, 2014.

13 Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE*, pages 144–153, Jun 2006.

14 Kathryn E. Gray. Safe cross-language inheritance. In *ECOOP*, pages 52–75, jul 2008.

15 Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA*, pages 231–245, Oct 2005.

16 Mathias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically composing languages in a modular way: Supporting C extensions for dynamic languages. In *Modularity*, March 2015.

17 Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *DLS*, pages 78–90, 2015.

18 Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *OOPSLA*, pages 318–326, Oct 1997.

19 Edgar T. Irons. Experience with an extensible language. *CACM*, 13(1):31–40, Jan 1970.

20 Gregory F. Johnson and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *POPL*, pages 141–151, Jan 1985.

21 Tomas Kalibera and Richard Jones. Quantifying performance changes with effect size confidence intervals. Technical Report 4-12, University of Kent, Jun 2012.

22 Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp Symb. Comput.*, 7(4):315–335, Dec 1994.

23 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *TOPLAS*, 31(3):12:1–12:44, Apr 2009.

24 James George Mitchell. *The design and construction of flexible and efficient interactive programming systems*. PhD thesis, Carnegie Mellon University, Jun 1970.

25 Melissa E. O'Neil. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation, 2015.

26 Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The full Monty. In *OOPSLA*, pages 217–232, 2013.

27 Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *SPE*, 23(5):529–566, 1993.

28 Aibek Sarimbekov, Andrej Podzimek, Lubomir Bulej, Yudi Zheng, Nathan Ricci, and Walter Binder. Characteristics of dynamic JVM languages. In *VMIL*, pages 11–20, Oct 2013.

29 Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *IVME*, pages 50–57, Jun 2003.

30 Kevin J. Sullivan, Mark Marchukov, and John Socha. Analysis of a conflict between aggregation and interface negotiation in Microsoft's component object model. *TOSE*, 25(4):584–599, Jul 1999.

31 Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, Oct 2008.

32 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Onward!*, pages 187–204, 2013.

33 Alexander Yermolovich, Christian Wimmer, and Michael Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *DLS*, pages 79–88, Oct 2009.

A Small Microbenchmarks

Section 7.1 outlined the small microbenchmarks used in our experiment. This appendix lists and describes each of the small microbenchmarks. The following microbenchmarks consist of an outer loop calling an inner function:

l1a0r The inner function takes an integer which is decremented to zero in a loop. Nothing is returned.
l1a1r The inner function takes an integer which is decremented to zero in a loop. After every decrement, the value is added to a sum total. The sum is returned.
ref_swap The inner function swaps its two arguments using references. Since Python has no support for references, there is no mono-Python variant of this benchmark.
return_simple The inner function returns a constant integer.
scopes The inner function takes a parameter and adds it to a variable from an outer scope. In the composed variants, the scope lookup crosses language boxes.
smallfunc The inner function takes three arguments a, b, and c, and returns $a + b * c$.
sum The inner function takes five arguments, sums them, and returns the result.
sum_meth As *sum*, except the sum is computed and returned by a method. The method belongs to an object which is allocated once and re-used.
sum_meth_attr As *sum_meth*, except that the result is stored to an attribute of the object.
total_list The inner function sums the elements of a list/array passed as an argument.

The following microbenchmarks consist of one function generating elements which another function consumes:

instchain A nested chain of objects is constructed and consumed in a loop. Each object in the chain has an attribute storing an integer. One function constructs the chain, another walks it summing the integers. These functions are called in a loop. In the composed variants, the outer loop is in one language and the construct and walk functions (including utility methods) are in the other.
lists One function constructs a list of integers, and another iterates over the list, summing its elements. These functions are called in a loop to repeatedly construct and sum lists. In the composed variant, the summing function is written in one language, with all other parts written in the other.
list_walk One function creates a linked list while the other function walks the list. Each element in the list is a three element tuple (x, y, n) where x and y are integers and n is a pointer to the next element, or the string `"end"` for the final element. As the list is walked, a counter is incremented by $y - x$. In the composed variant, the list creation and walking functions are in a different language from the outer loop.

The l1a0r, l1a1r, lists, smallfunc, and list_walk microbenchmarks are ports of benchmarks from [3]. All other small microbenchmarks were created specifically to test PyHyp.

Table 4 Absolute microbenchmark timings.

Benchmark	CPython	HHVM	HippyVM	PyHyp$_{\text{PHP}}$	PyHyp$_{\text{Py}}$	PyHyp$_{\text{mono}}$	PyPy	Zend
instchain	11.323 ±0.0208	3.232 ±0.0016	0.309 ±0.0002	0.338 ±0.0003		0.378 ±0.0002	0.228 ±0.0001	12.345 ±0.0519
l1a0r	15.965 ±0.0011	0.752 ±0.0003	0.254 ±0.0000	0.186 ±0.0000	0.252 ±0.0000	0.252 ±0.0000	0.249 ±0.0019	7.198 ±0.0005
l1a1r	16.473 ±0.0162	0.586 ±0.0001	0.257 ±0.0000	0.197 ±0.0002	0.256 ±0.0000	0.256 ±0.0000	0.224 ±0.0003	7.689 ±0.0166
lists	3.871 ±0.0021	0.448 ±0.0015	0.469 ±0.0005	0.481 ±0.0008	0.269 ±0.0003	0.470 ±0.0005	0.239 ±0.0002	7.036 ±0.0136
ref_swap		2.573 ±0.0001	0.306 ±0.0001	0.306 ±0.0000	0.215 ±0.0000	0.306 ±0.0000		16.343 ±0.0008
return_simple	27.678 ±0.0273	1.767 ±0.0005	0.251 ±0.0000	0.251 ±0.0000	0.195 ±0.0000	0.251 ±0.0000	0.223 ±0.0000	21.239 ±0.0161
scopes	17.854 ±0.0065	2.009 ±0.0002	0.603 ±0.0003	0.134 ±0.0000	0.124 ±0.0001	0.601 ±0.0002	0.134 ±0.0000	20.412 ±0.0014
smallfunc	46.912 ±0.0368	3.278 ±0.0002	0.251 ±0.0000	0.251 ±0.0000	0.188 ±0.0000	0.251 ±0.0000	0.251 ±0.0000	57.862 ±0.0025
sum	23.612 ±0.0201	1.440 ±0.0000	0.074 ±0.0000	0.074 ±0.0000	0.056 ±0.0000	0.074 ±0.0000	0.065 ±0.0000	31.124 ±0.0063
sum_meth	25.428 ±0.0994	1.793 ±0.0021	0.074 ±0.0000	0.074 ±0.0000		0.074 ±0.0000	0.065 ±0.0000	33.283 ±0.0360
sum_meth_attr	31.930 ±0.0046	4.351 ±0.0003	0.243 ±0.0003	0.243 ±0.0014		0.275 ±0.0001	0.220 ±0.0003	35.305 ±0.0125
total_list	8.076 ±0.0058	0.943 ±0.0003	0.363 ±0.0001	0.420 ±0.0001	0.633 ±0.0001	0.360 ±0.0002	0.246 ±0.0001	14.138 ±0.0257
walk_list	1.054 ±0.0007	0.085 ±0.0000	0.162 ±0.0001	0.208 ±0.0003	0.333 ±0.0003	0.210 ±0.0003	0.225 ±0.0001	2.218 ±0.0144
deltablue	0.901 ±0.0006	36.609 ±0.0320	0.236 ±0.0005	0.055 ±0.0002		0.246 ±0.0005	0.025 ±0.0001	7.860 ±0.1417
fannkuch	15.273 ±0.0168	2.480 ±0.0017	1.371 ±0.0004	0.742 ±0.0002	1.403 ±0.0002	1.393 ±0.0002	0.746 ±0.0002	10.676 ±0.0092
mandel		0.460 ±0.0033	0.536 ±0.0003	0.581 ±0.0002	0.287 ±0.0000	0.581 ±0.0001		4.211 ±0.0112
richards	11.901 ±0.0055	5.263 ±0.0027	0.377 ±0.0004	0.442 ±0.0002		0.392 ±0.0003	0.216 ±0.0002	10.709 ±0.0091